Comp151

Overloading Operators & Container Classes

Container Classes

- Container classes are a typical use for class templates, since we need container classes for objects of many different types, and the types are not known when the container class is designed.
- Using the tools learnt from **IntArray** we will design a container that looks like an array. Doing this will show us how the C++ *vector, list, etc.* are actually defined.
- We want the container to be *homogeneous*: all the elements must have the same type.
- But should a container with 10 `int` elements be the same type as a container with 20 `int` elements?

  Both choices are sensible design decisions.

Remark: The `vector` type in STL is better, so this is just for our understanding.

Example: Container Class — bunch.h

```
template <typename T, int N>
class Bunch
{
  private:
    T _value[N];

  public:
    Bunch();
    Bunch(const Bunch &B);
    ~Bunch();
    int size() const { return N; }
    T& operator[ ](int i) { return _value[i]; }
    T & operator=(const Bunch &B)
};
```

Example: Use of Class Bunch


```
Bunch<int, 10> A;
cout << A[3];
A[7] = 13;

Bunch<string, 50> B;
B[49] = "Hello world";

Bunch<string, 50> C;
C = B;                                  // Legal

Bunch<int, 20> D;
D = A;          // Error: D and A are of different types
```

A More Flexible Container Class — array.h


```
template <typename T>
class Array
{
  private:
    T* _value;
    int _size;
  public:
    Array(int n = 10);                              // Default constructor
    Array(const Array & A);                         // Copy constructor
    ~Array();

    int size() const { return _size; }
    Array & operator=(const Array &A);              // Assignment operator
    T& operator[ ](int i) { return _value[i]; };    // Access to an element
 };
```

Example: Use of Class Array

```
#include <iostream>
#include "array.h"

int main()
{
    Array<int> a;
    cout << a.size() << endl;
    a[9] = 17;
    cout << a[2] << endl;

    Array<int> b(5);
    cout << b.size() << endl;

    Array<int> c(20);
    c[1] = 5;
    cout << c[1] << endl;

    a = c;
    cout << a[2] << endl;
}
```

Example: Constructors/Destructor of Class Array

```
template <typename T>
Array<T>::Array(int n) : _value( new T[n] ), _size (n) { }
```

```
template <typename T>
Array<T>::Array(const Array<T> &A)
    : _value( new T[A._size] ), _size (A._size)
{
    for (int i = 0; i < _size; ++i)
        _value[i] = A._value[i];
}
```

```
template <typename T>
Array<T>::~Array() { delete [ ] _value; }
```

Assignment Operator ..

Idea: To assign `b = a`, first throw away the old data `b._value`, then create a new one and assign the elements from `a._value`.

When the assignment argument is the same as the object being assigned to, don't do anything.

```
template <typename T>
Array<T>& Array<T>::operator=(const Array<T>& A)
{
    if (this != &A) {
        delete [ ] _value;
        _size = A._size;
        _value = new T[_size];

        for (int i = 0; i < _size; ++i)
            _value[i] = A._value[i];
    }
    return *this;
}
```

Output Operator

- The following output operator is not a member of the `Array<T>` class, but a function template.
- Function templates and class templates work together very well: We can use function templates to implement functions that will work on any class created from a class template.

```
template <typename T>
ostream& operator<<(ostream& os, const Array<T>& A)
{
    for (int i = 0; i < A.size(); ++i)
        os << A[i] << ' ';

    return os;
}
```